KMeans Implementation

Parallel Computing Course Project

Federico Nocentini, Corso Vignoli

Supervisor: Prof. Marco Bertini

Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Firenze

INDEX

Introduction

K-means clustering is one of the simplest and most popular unsupervised machine learning algorithms.

Keywords :

- Cluster
- Centroid
- "Means"

It works only in Euclidean Spaces (in our case 2D-space).

Formal Definition

Given an initial set of observations in a d-dimensional Euclidean space,

Given a set of K random centroids,

Assignment Step : $C_i^{(t)} = \left\{ x_p : \|x_p - c_i^{(t)}\|^2 \leq \|x_p - c_j^{(t)}\|^2, \forall j : 1 \leq j \leq K \right\}$

Update Step : $c_i^{(t+1)} = \frac{1}{|C_i^{(t)}|} \sum_{x_j \in C_i^{(t)}} x_j$

The algorithm converges when assignments and centroids no longer change.

Proposed Approach

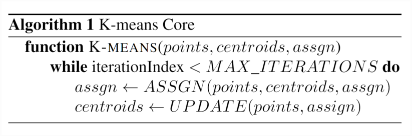
In this work we propose 3 different implementations of the algorithm:

1. A sequential version with **Python**;
2. A parallel version with **OpenMP**;
3. A parallel version with **CUDA**.

Sequential implementation : Python

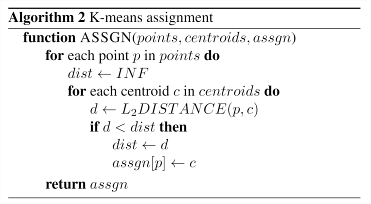
It is a simple translation of the principle behind the K-means in Python.

The MAX-ITERATIONS constant represents the number of iterations of the algorithm.



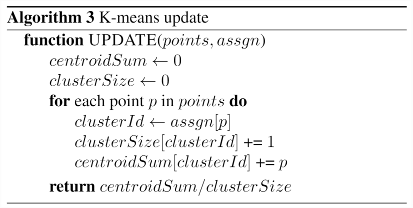
Algorithm 1 K-means Core

```
function K-MEANS($points, centroids, assgn$)
    while iterationIndex $< MAX\_ITERATIONS$ do
        $assgn \leftarrow ASSGN(points, centroids, assgn)$
        $centroids \leftarrow UPDATE(points, assign)$
```



Algorithm 2 K-means assignment

```
function ASSGN($points, centroids, assgn$)
    for each point $p$ in $points$ do
        $dist \leftarrow INF$
        for each centroid $c$ in $centroids$ do
            $d \leftarrow L_2DISTANCE(p, c)$
            if $d < dist$ then
                $dist \leftarrow d$
                $assgn[p] \leftarrow c$
    return $assgn$
```

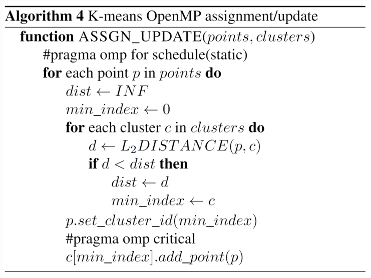


Algorithm 3 K-means update

```
function UPDATE($points, assgn$)
    $centroidSum \leftarrow 0$
    $clusterSize \leftarrow 0$
    for each point $p$ in $points$ do
        $clusterId \leftarrow assgn[p]$
        $clusterSize[clusterId]$ += 1
        $centroidSum[clusterId]$ += $p$
    return $centroidSum/clusterSize$
```

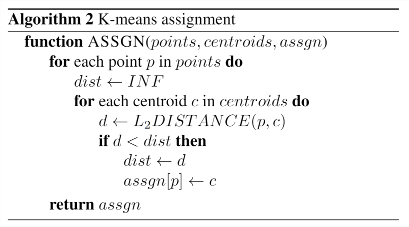
Parallel Implementation : OpenMP

The OpenMP library lets us transform the sequential version into a parallel one with several **pragma** directive.



Algorithm 4 K-means OpenMP assignment/update

```
function ASSGN_UPDATE($(points, clusters)$)
    #pragma omp for schedule(static)
    for each point $p$ in $points$ do
        $dist \leftarrow INF$
        $min\_index \leftarrow 0$
        for each cluster $c$ in $clusters$ do
            $d \leftarrow L_2DISTANCE(p, c)$
            if $d < dist$ then
                $dist \leftarrow d$
                $min\_index \leftarrow c$
        $p.set\_cluster\_id(min\_index)$
        #pragma omp critical
        $c[min\_index].add\_point(p)$
```



Algorithm 2 K-means assignment

```
function ASSGN($(points, centroids, assgn)$)
    for each point $p$ in $points$ do
        $dist \leftarrow INF$
        for each centroid $c$ in $centroids$ do
            $d \leftarrow L_2DISTANCE(p, c)$
            if $d < dist$ then
                $dist \leftarrow d$
                $assgn[p] \leftarrow c$
    return $assgn$
```

Exploiting GPUs: CUDA

Implementing the algorithm with CUDA lets us take advantage of the great number of cores of GPUs.

The processing of each point can be assigned to a different thread.

The N points to be clustered are stored in 2 arrays of one dimension, which contain the features of each point, respectively:

$$p_x = [x_1, x_2, \ldots, x_N]\,,\, p_y = [y_1, y_2, \ldots, y_N]$$

The same data organization is applied to centroids:

$$c_x = [x_1, x_2, \ldots, x_K]\,,\, c_y = [y_1, y_2, \ldots, y_K]$$

Parallel Implementation : CUDA

Algorithm 5 K-means Core CUDA

```
function K-MEANS($p_x, p_y, c_x, c_y, assgn, c\_size$)
    while iterationIndex $< MAX\_ITERATIONS$ do
        C_ASSGN($p_x, p_y, c_x, c_y, assgn$)
        $sum_x \leftarrow 0$
        $sum_y \leftarrow 0$
        $c\_size \leftarrow 0$
        C_UPDATE($p_x, p_y, sum_x, sum_y, assgn, c\_size$)
        $c_x \leftarrow sum_x / c\_size$
        $c_y \leftarrow sum_y / c\_size$
```

Algorithm 6 K-means Assignment CUDA

```
function C_ASSGN($p_x, p_y, c_x, c_y, assgn$)
    $idx \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
    if $idx > N$ then
        return
    $dist \leftarrow INF$
    $closest\_centroid \leftarrow 0$
    $c \leftarrow 0$
    while $c < K$ do
        $d \leftarrow DISTANCE(p_x[idx], p_y[idx], c_x[c], c_y[c])$
        if $d < dist$ then
            $dist \leftarrow d$
            $closest\_centroid \leftarrow c$
        $c \leftarrow c + 1$
    $assgn[idx] \leftarrow c$
```

Algorithm 7 K-means Update CUDA

```
function C_UPDATE($p_x, p_y, sum_x, sum_y, assgn, c\_size$)
    $idx \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
    if $idx > N$ then
        return
    $cluster\_id \leftarrow assgn[idx]$
    $sum_x[cluster\_id] \leftarrow sum_x[cluster\_id] + p_x[idx]$
    $sum_y[cluster\_id] \leftarrow sum_y[cluster\_id] + p_y[idx]$
    $c\_size[cluster\_id] \leftarrow c\_size[cluster\_id] + 1$
```

Experimental Results I

The performances are compared with the speedup metric, computed as:

$$S = \frac{t_S}{t_P}$$

The tests have been executed on a machine with:

- CPU: Intel® Core™ i7-10710U, 6 Cores/12 Threads
- GPU: GeForce GTX 1650 Mobile / Max-Q 4GB with CUDA 10.1

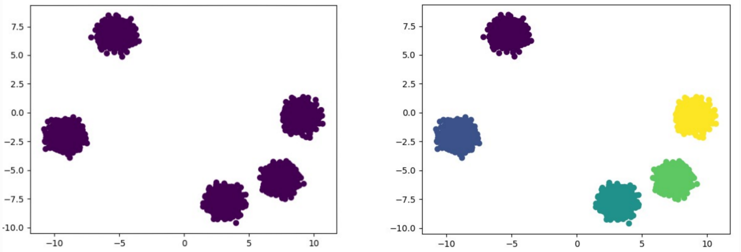


Figure 1. Example of a dataset with 10000 points respectively not clustered and clustered

Experimental Results II

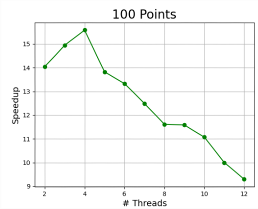
The datasets used to evaluate the different implementations have been generated with the *make_blob()* function of *sklearn.datasets*.

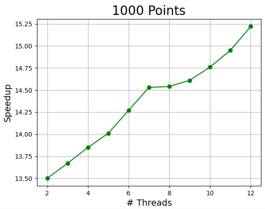
They are gaussian distributions with 5 centers and standard deviation equal to 0.5 and are composed by respectively 100, 1000, 10000, 100000 and 1000000 2D points.

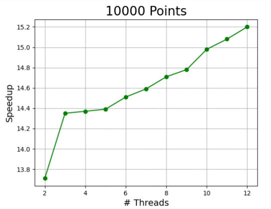
The *MAX_ITERATIONS* constant has been set to 20 because it has been estimated empirically that 20 iterations are enough to make all the centroids converge.

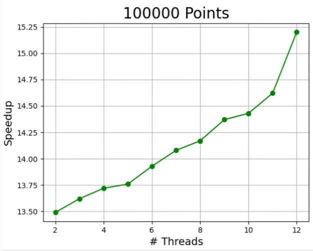
OpenMP

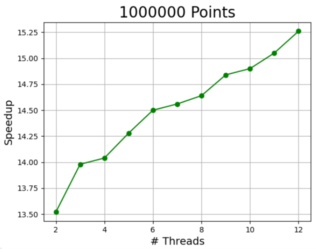
To evaluate the performances of the OpenMP implementation, it has been executed on each dataset with an increasing number of threads.





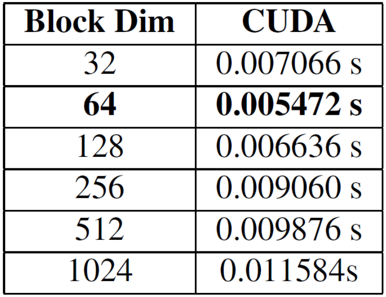






CUDA I

To evaluate our CUDA implementation, we test the block dimension with fixed dataset dimension, 10000 points.

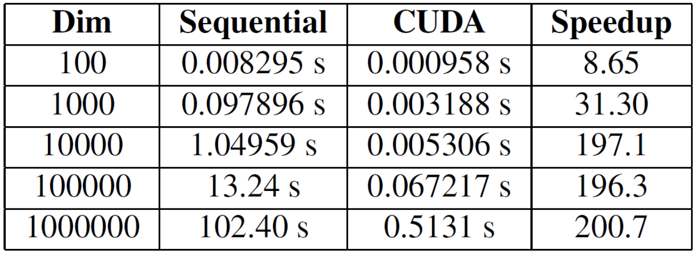


Block Dim	**CUDA**
32	0.007066 s
64	**0.005472 s**
128	0.006636 s
256	0.009060 s
512	0.009876 s
1024	0.011584s

Table 7. Execution time for 10000 points dataset while changing the block dimension (best result in bold)

CUDA II

To evaluate the performances of the CUDA implementation, it has been executed on each dataset like the OpenMP implementation with block dimension fixed at 64.



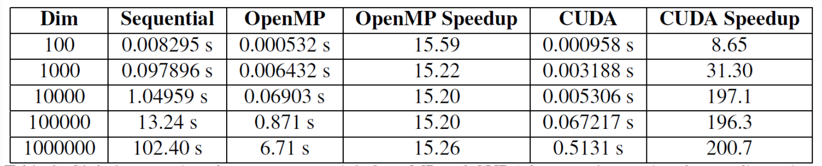
Dim	Sequential	CUDA	Speedup
100	0.008295 s	0.000958 s	8.65
1000	0.097896 s	0.003188 s	31.30
10000	1.04959 s	0.005306 s	197.1
100000	13.24 s	0.067217 s	196.3
1000000	102.40 s	0.5131 s	200.7

Table 8. Execution time and speedup obtained with sequential and CUDA implementations for increasing dataset dimension

Global Comparison

The CUDA algorithm **abundantly outperforms** both the sequential and the OpenMP ones, at the expense of a more complicated implementation.

OpenMP, instead, lets us to achieve a **noticable speedup** with just several directive.



Dim	Sequential	OpenMP	OpenMP Speedup	CUDA	CUDA Speedup
100	0.008295 s	0.000532 s	15.59	0.000958 s	8.65
1000	0.097896 s	0.006432 s	15.22	0.003188 s	31.30
10000	1.04959 s	0.06903 s	15.20	0.005306 s	197.1
100000	13.24 s	0.871 s	15.20	0.067217 s	196.3
1000000	102.40 s	6.71 s	15.26	0.5131 s	200.7

Table 3. Global comparison between sequential, OpenMP and CUDA best results varying dataset dimension

Conclusions

The K-means clustering algorithm has an embarassingly parallel structure suitable for parallel computing.

A parallel implementation with **OpenMP** allows to obtain a speedup equal to more than **15**.

A parallel **CUDA** implementation allows to obtain a speedup up to **200**.

So it's far more convenient to implement this algorithm in CUDA, due to the higher number of cores of the GPU.